1988

Semantic shortcomings of database management systems based on a relational model.

Wall, Jonathan S.

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

SEMANTIC SHORTCOMINGS OF
DATABASE MANAGEMENT SYSTEMS
BASED ON A RELATIONAL MODEL

by

Jonathan S. Wall

June 1988

Thesis Advisor: C. Thomas Wu

Approved for public release; distribution is unlimited

REPORT DOCUMENTATION PAGE

a. REPORT SECURITY CLASSIFICATION Unclassified	1b RESTRICTIVE MARKINGS
a. SECURITY CLASSIFICATION AUTHORITY	3 DISTRIBUTION / AVAILABILITY OF REPORT Approved for public release; Distribution is unlimited
b. DECLASSIFICATION / DOWNGRADING SCHEDULE	
PERFORMING ORGANIZATION REPORT NUMBER(S)	5 MONITORING ORGANIZATION REPORT NUMBER(S)
a. NAME OF PERFORMING ORGANIZATION Naval Postgraduate School 6b OFFICE SYMBOL (*If applicable*) Code 52	7a NAME OF MONITORING ORGANIZATION Naval Postgraduate School
c. ADDRESS (*City, State, and ZIP Code*) Monterey, California 93943-5000	7b. ADDRESS (*City, State, and ZIP Code*) Monterey, California 93943-5000
a. NAME OF FUNDING / SPONSORING ORGANIZATION 8b OFFICE SYMBOL (*If applicable*)	9 PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
c. ADDRESS (*City, State, and ZIP Code*)	10 SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO / TASK NO / WORK UNIT ACCESSION NO

11. TITLE (*Include Security Classification*)

SEMANTIC SHORTCOMINGS OF DATABASE MANAGEMENT SYSTEMS BASED ON A RELATIONAL MODEL

12. PERSONAL AUTHOR(S)
Wall, Jonathan S.

13a. TYPE OF REPORT	13b TIME COVERED	14. DATE OF REPORT (*Year, Month, Day*)	15 PAGE COUNT
Master's Thesis	FROM _____ TO _____	1988 June	97

16. SUPPLEMENTARY NOTATION
The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

17 COSATI CODES			18 SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*)
FIELD	GROUP	SUB-GROUP	Relational database management systems; Attributes; Joins

19. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

To many critics and researchers, semantic meagerness is the main limitation of relational DBMS. The burden placed on users to remember attribute names and their domains is discussed and difficulties associated with the lack of set as a type for an attribute is examined. The paper explores the implications to high-level query languages necessitated by a set-type attribute. The allowance of semantically improper joins by DBMS is studied as is the lack of strong data type checking. The semantic shortcomings of system-chosen access paths is discussed. These problems are followed by recommended solutions.

20 DISTRIBUTION / AVAILABILITY OF ABSTRACT	21 ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL Prof. C. Thomas Wu	22b TELEPHONE (*Include Area Code*) (408) 646-3391 22c OFFICE SYMBOL Code 52Wq

Approved for public release; distribution is unlimited.

SEMANTIC SHORTCOMINGS OF DATABASE MANAGEMENT SYSTEMS BASED ON A RELATIONAL MODEL

by

Jonathan S. Wall
Lieutenant, United States Navy
B.S., United States Naval Academy, 1981

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL

June 1988

ABSTRACT

To many critics and researchers, semantic meagerness is the main limitation of relational DBMS. The burden placed on users to remember attribute names and their domains is discussed and difficulties associated with the lack of set as a type for an attribute is examined. The paper explores the implications to high-level query languages necessitated by a set-type attribute. The allowance of semantically improper joins by DBMS is studied as is the lack of strong data type checking. The semantic shortcomings of system-chosen access paths is discussed. These problems are followed by recommended solutions.

THESIS DISCLAIMER

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

TABLE OF CONTENTS

LIST OF FIGURES

I. INTRODUCTION

Database technology is one of the most rapidly growing fields in computer science. Its popularity among corporations and government agencies within the last ten years is staggering. Database systems on personal computers is now commonplace.

Database denotes collections of data shared by end-users of computer systems. It is the most modern technique of data storage hence its popularity and importance. Decision-makers within an organization make decisions by accessing the database. Obviously, the ease of access, security, and integrity of the database is extremely important.

Database systems are distinguished from other systems by certain features. The following may be considered as constituting the major features of a database system [Ref. 1:p. 61]:

- performance optimization
- concurrent usage
- data protection
- data independence
- flexibility of data structure.

A software program, known as the database management system (DBMS) manages the database. The DBMS controls the storing and retrieval of data, and the users

themselves. DBMS have also facilitated the development of many database applications (computer applications where many users at terminals concurrently access a database).

In the coming years, database systems will become increasingly important. With the cost of labor steadily increasing and the cost of computers decreasing, people are being replaced by computers. [Ref. 2:p. 1] There is strong reason to believe this trend will continue.

There are currently three commonly implemented database models: 1) relational database, 2) hierarchical database, and 3) network database. Relational DBMSs are based on a strong theoretical foundation which is in contrast to the network structure which was borrowed from telecommunications and the hierarchical structure which was borrowed from bill of material systems. [Ref. 3:p. 56] Other features which attract users to relational DBMS are its ease of use, data independence, and table data structure. Relational DBMS have only recently begun to gain widespread popularity in the commercial environment. The relational system has been touted as the system of the future. Whether or not it lives up to this billing remains to be seen. The primary goal of this thesis is to present problems and shortcomings of DBMS based on the relational model and to provide recommendations and solutions to these problems.

Chapter II describes the basic concepts of a database. The chapter gives a definition of a database system, describes the components of a database system, and lists the advantages and disadvantages of database processing. Chapter III describes the relational database model. A relation is defined and common relational database model terms are explained. The traditional relational algebra operators are detailed as are the special relational operators. Finally, relational data manipulation languages are discussed. Chapter IV describes the advantages of the relational approach to database management systems. Advantages such as ease of understanding and data independence, as well as others, are outlined. Chapter V examines the problems of the relational approach to database management systems: Semantic burdens on the user, the lack of set-type attributes, query language modifications required by set-type attributes, semantically improper joins, the lack of strong data type checking, and access paths. The problems are detailed and followed by recommended solutions. Finally, conclusions are presented in Chapter VI.

II. DATABASE CONCEPTS

A. DEFINITION OF A DATABASE SYSTEM

A database system is a system whose overall purpose is to maintain information and to make that information available on demand. The information is whatever the individual or organization deems to be significant to the entity being served. In short, information is anything required by the individual or organization to help in the process of making decisions. [Ref. 4:p. 4] A database may also be considered a collection of facts or a repository for stored data which is both integrated and shared. Integrated means the database may be considered a consolidation of several otherwise discrete data files. Any redundancy amongst the files is either fully or partially eliminated. Shared means that separate pieces of data in the database may be shared among many different users. Each user may have access to the same piece of data and different users may use that data for different purposes. Different users may also access the data at the same time, known as "concurrent access". [Ref. 4:p. 5]

The definition can be summarized with the following points:

- A database is a generalized compilation of data.
- This compilation is integrated to reduce redundancy.

- Data structure is based on natural data relationships which provides all necessary access paths. The required access path to a unit of data is really a result of its relationship to other data. The ability to represent the natural data relationships with all the necessary access path is the essence of the distinction between a database and a conventional file.

- A database must supply the differing data needs of users in an efficient and effective manner. [Ref. 1:p. 5]

The importance of database systems to today's organizations cannot be understated. A database is a shared resource, thus its design and use must be managed with all the users in mind. The difference between sound decisions and poor decisions many times rests with the quality of information in the database.

The person who controls the database in an organization is known as the Database Administrator (DBA). This position is critical because this person (or group of persons) assumes responsibility for protecting the database while at the same time attempting to maximize benefits to users. [Ref. 2:p. 536] The DBA's responsibilities include the following:

- determines the information content of the database
- determines the storage structure and access strategy
- provides a liaison with users
- specifies security and integrity checks
- defines backup and recovery strategy
- monitors performance and responds to changing requirements. [Ref. 4:pp. 25-27]

The final piece of the puzzle is the database management system (DBMS). The DBMS is the software that manages a database; the word "management" may be interpreted to include the functions of creation and maintenance. [Ref. 1:p. 6] A full scale DBMS provides the following capabilities:

- storage, update, and retrieval of data
- a catalog accessible to users which provides data description
- transaction support to ensure that all or none of a series of database changes are reflected in the relevant databases
- recovery services in the event of a failure (system or program)
- concurrency control services to ensure that concurrent transactions function the same way as if having been run in some sequential order
- authorization services to ensure that access to and manipulation of data is in accordance with defined constraints on users and programs
- integration that includes support for data communication
- integrity services that ensure database states and changes in this state conform to specified rules. [Ref. 5:p. 114]

The database system provides the organization with centralized control of its operational data. [Ref. 4:p. 9] Without a database system, an organization is subject to a wide range of private files interfacing with applications. Control of one of any organization's most important assets, the operational data, is shaky. This

may prove to be very hazardous to the corporation's well-being.

B. COMPONENTS OF A DATABASE SYSTEM

A database system consists of four major components: data, hardware, software, and users. Figure 2.1 shows an example of the arrangement of a simplistic system.

1. Data

As stated earlier, a database is a collection of integrated files. Kroenke states that "a database is a collection of files and relationships among records in those files." [Ref. 2:p. 11]

Holding true to the lexicon in the computer industry, bits are grouped into bytes (8 bits = 1 byte) or characters, characters are grouped into fields, and fields are grouped into records. A collection of records is called a file. [Ref. 2:p. 11]

Database processing differs significantly from file processing where each file is considered to exist independently and the structure of the files is distributed across application programs.

For tutorial purposes, we will assume there is just one database, containing the totality of all stored data in the system although normally the system is split into one or more databases.

2. Hardware

There is no special hardware needed for database systems. Hardware consists of device controllers, input/output channels, secondary storage devices (disks, drums, etc.) on which the database resides, together with associated devices. It is assumed primary storage will not be large enough store the entire database.

Database applications often require extensive resources (i.e., larger main memory, faster central processing unit, and more direct access storage). This can be quite expensive. Database processing also involves special programs and overhead data.

Special purpose computers that perform database processing functions, called database machines, were announced by several vendors in 1982 [Ref. 2:p. 8] According to this type of architecture, the computer processing the application program sends requests for service and data over a channel to the database machine. The machine processes the requests and sends results, data, or messages back to the main computer. In this manner, database processing can be performed in a concurrent manner with applications processing. The effectiveness of database machines remains to be seen.

3. Software

The layer of software between the physical database (i.e., the data as actually stored) and the

users of the system is known as the database management system (DBMS). Requests from users to interact with the database are processed by the DBMS. The DBMS acts as a safeguard between the users and hardware level details.

The operating system (OS) is a program which controls the computer's resources thus relieving users of this burden. Operating system programs cause tasks to be performed and may be considered the nucleus of all the other programs.

The Communications Control Program (CCP) performs communication-oriented tasks. It provides communications error checking (and correction if errors are found), coordinates terminal activity, sends messages to their proper destination, and formats messages for various types of terminal equipment.

Application programs (AP) are computer applications where many users at terminals concurrently access a database and are tailored to specific business needs. Specific needs such as order entry, inventory accounting, and billing are satisfied. [Ref. 2:p. 9]

4. Users

Three broad classes of users interface with the database system: application programmers, end-users and the database administrator (DBA).

The application programmer is responsible for writing application programs that use the database.

Typically written in a language such as COBOL or PL/1, application programs are used with data for retrieving information, creating new information, and deleting or changing existing information. Such functions are performed by the DBMS after it receives the appropriate request. The programs themselves may be conventional batch applications or on-line applications which functions to support an end-user who accesses the database from an on-line terminal.

An end-user may employ a query language provided as a composite part of the system to perform the functions of retrieval, creation, deletion, and modification of data. The alternative is for the end-user to utilize one of the on-line application programs that accepts commands from the terminal and then issues requests to the DBMS on the end-user's behalf.

The database administrator (DBA) mentioned earlier is the person (or group of persons) responsible for the control of the database system. The DBA staff serves as a guardian of the database and as a focal point for resolving users' conflicts.

C. ADVANTAGES AND DISADVANTAGES OF DATABASE PROCESSING

As mentioned earlier, an advantage of a database system is that it provides the organization with centralized control of its operational data [Ref. 4:p. 9]. Also, database processing allows more information to

be yielded from a given amount of data. Information is then gained by processing these recorded facts and figures, the data. Centralized control of the operational data provides the following advantages [Ref. 4:pp. 10-12].

1. Redundancy Can Be Reduced

One important advantage of database processing is the elimination or reduction of data duplication. In conventional file processing systems, each application has its own private files. This may lead to considerable redundancy in stored data therefore wasting storage space.

Elimination of duplicated information saves file space and may reduce processing requirements. It should be noted that all redundancy should not necessarily be eliminated. However, redundancy that does exist must be carefully monitored. [Ref. 2:pp. 3-6]

2. Avoidance of Inconsistency

This follows closely with the above point. If two different entries in the database represent a single fact about the real world, there will undoubtedly be occasions where the two entries do not agree (i.e., when only one entry has been properly updated). At such times the database is said to be inconsistent. A database in an inconsistent state may supply users with incorrect or conflicting information which leads to user distrust of

computer generated output. If redundancy is controlled, the system can ensure that the database will never be inconsistent in the eyes of the user, by guaranteeing that any change made to either of the two entries is automatically applied to the other one also. This process is known as "propagating updates." [Ref. 2:pp. 3-6]

3. Shared Data

As discussed earlier, sharing means existing as well as new applications that are developed can share the data in the database. Users have access to the same data and different users may use the data in different ways and for different purposes. [Ref. 2:pp. 3-6]

4. Enforcements of Standards

Because the data is centralized (vice private files for each application), the DBA is able to ensure that all pertinent standards are observed with respect to the representation of the data. Applicable standards may include any or all of the following: organization, state, division, industry, national, and international standards. Standardizing helps in data migration between systems. [Ref. 2:pp. 3-6]

5. Application of Security Restrictions

With complete control over the database system, the DBA can ensure the database may be accessed only through the proper channels therefore requiring security

checks to be carried out whenever access to classified data is attempted. It should be noted that the very nature of a centralized database system requires a sound security system be in place. [Ref. 2:pp. 3-6]

6. Data Integrity

Data integrity deals with the problem of guaranteeing the data in the database is accurate. There is a lack of data integrity if two entries that represent the same fact are inconsistent (which can only occur if redundancy exists in the stored data). Because the database is shared (unlike private files), data integrity is of extreme importance. Most current database products are weak in this area. A common result from this shortcoming is conflicting reports by corporation employees who use these database products. [Ref. 2:pp. 3-6]

7. Balancing of Conflicting Reports

Because the database system is shared by the whole organization as opposed to just individuals, the DBA must structure the system to provide service that is in the best interests of that organization. [Ref. 2:pp. 3-6]

There are many disadvantages that come along with databases. A major disadvantage of database processing is that it can be very expensive. DBMS are not inexpensive, and because of its main memory requirements,

additional memory may have to be purchased. The system may also gain exclusive control of the Central Processing Unit (CPU) forcing the user to upgrade to a more powerful computer.

Data processing tends to be very complex. The database system and application programs must be able to process large amounts of data that are interrelated in different formats. This results in sophisticated programming and requires highly trained programming and maintenance personnel. Backup and recovery of data also increases system complexity and is difficult to carry out in the database environment.

Another disadvantage is that integration, and hence centralization, increases vulnerability. The entire system may fail if one component fails. Obviously, this becomes extremely critical if the users' organization depends heavily upon the database for its day-to-day operations. [Ref. 2:pp. 6-7]

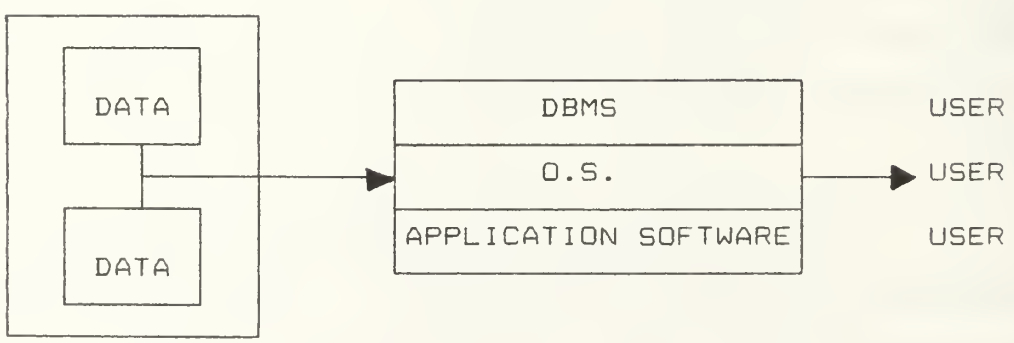

Figure 2.1. Simple View of Database System

III. RELATIONAL DATABASE MODEL

A. RELATIONAL DATA STRUCTURE

The relational model was first proposed by Dr. E. F. Codd in 1970. Only recently has the model moved from the world of theoretical interest to that of practical significance. This was the result of the announcement of several important relational DBMS products (i.e., SQL/DS, ORACLE, etc.).

A relation is a mathematical term for a two-dimensional table. It is characterized by rows and columns that contain data item values. It is called a relation and not a matrix due to a lack of homogeneity in its entries; the entries are homogeneous in the columns, but not in the rows. A relational database is composed of such relations which may be stored on a physical device in a variety of ways. [Ref. 1:p. 130]

To explain the relational data structure, sample data in relational form will be very helpful. Figure 3.1 shows a relational view of data which is organized into three tables: STUDENT, TEACHER, and STUDENT-TEACHER (ST). The STUDENT table contains, for each student, a student identification number, a student name, the year in school of that student, and the district where the student attends school. The TEACHER table contains, for each teacher, a teacher identification number, a course

name, a class size, the credit hours for the course, and the teaching district. As in the STUDENT relation, we assume each teacher has a unique identification number. The STUDENT-TEACHER table contains a student identification number, a teacher identification number, and a tenure value.

1. Definition of a Relation

Given a collection of sets D1,D2,...,Dn (not necessarily distinct) R is relation on those n sets if it is a set of ordered n-tuples <d1,d2,...,dn> such that d1 belongs to D1, d2 belongs to D2,..., dn belongs to Dn. Sets D1, D2, ...,Dn are the domains of R. The value n is the degree of R. Relations of degree 1 are called unary; degree 2:binary, degree 3:ternary, and degree n:n-ary. [Ref. 4:p. 83]

An equivalent definition of a relation can be given from a mathematical set-theory perspective. A relation is any subset of the Cartesian product of one or more domains. For example, if we have n sets, say n = 2, A1 = {a,b} and A2 = {1,2,3}, then A1 x A2 is the Cartesian product of these n sets. It is the set of all possible ordered n-tuples <a1,a2> such that a1 belongs to A1 and a2 belongs to A2. The result of A1 x A2 is {(a,1),(a,2),(a,3),(b,1),(b,2),(b,3)}. Figure 3.2 shows the Cartesian product of two sets SID and TID (student and teacher identification number).

STUDENT

SID	SN	YR	DISTRICT
S1	ROSE	1	MTY
S2	STONE	2	SAL
S3	VENTURA	3	SAL

(a)

TEACHER

TID	COURSE	CLSIZE	CREDIT	DISTRICT
T1	HIST	MED	2	MTY
T2	MATH	LRG	4	SAL
T3	ECON	SML	3	CML
T4	PSYCH	MED	3	MTY

(b)

ST

SID	TID	TENURE/AMOUNT
S1	T1	3
S1	T2	2
S1	T3	4
S2	T1	3
S2	T2	4
S3	T2	2

(c)

Figure 3.1. Student, Teacher and Student/Teacher Relations

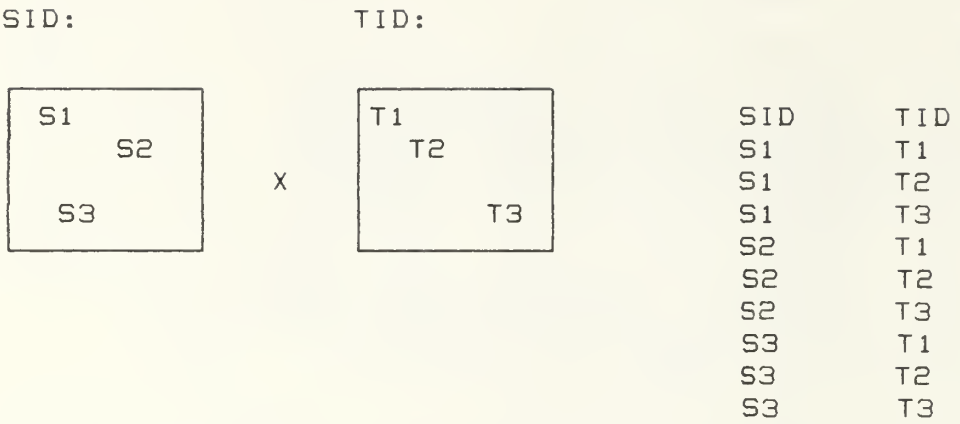

Figure 3.2. Example of Cartesian Product

A relation called STUDENT of degree four is illustrated in Figure 3.1(a). The four domains are sets of values representing student identification number (SID), student name (SN), the year in school of each student (YR), and the district of the school (DISTRICT) respectively. The YR domain, for example, is the set of all valid year numbers. Note that there may be some year numbers included in this domain that do not actually appear in the STUDENT relation at this particular time (relations are time-varying).

A relation may be viewed as a table, where each row is called a tuple (corresponding to records) and each column represents a field within the record. The columns are called attributes. The number of tuples in a relation is called the cardinality of that relation. For

example, the cardinality of the STUDENT relation is three and it has four attributes (columns).

A domain may be thought of as a bank of values from which are drawn the actual values for a given attribute. Keep in mind that a relation is a set and sets are not ordered. There is no ordering defined among the tuples of a relation. However, the domains of a relation do have an ordering among them. If we have a tuple (a1,a2,...,an) with n components, the value of the j th component in this n-tuple must be pulled from the j th domain. [Ref. 4:p. 85] In Figure 3.2(a), the third tuple of the STUDENT relation is (S3, VENTURA, 3, SAL). The value of the third field of this tuple under the attribute YR is drawn from the third domain all positive integers. In mathematical terms, rearranging the four columns results in a different relation.

It is important to understand the difference between a domain [Ref. 4:p. 85] and attributes which are drawn from that domain. Figure 3.3 shows a part of a relational schema in which four domains (STUDENT_ID, STU_NAME, YEAR, and LOCATION) and one relation (STUDENT) are defined using a data definition language. The relation is declared with four attributes (SID, SN, YR, and DISTRICT), and each attribute is designated as being pulled from a corresponding domain. It may be the case that more than one attribute has the same domain. Data

values from attributes having the same domain are what allow the relational system to associate tuples from different relations.

```
DOMAIN      STUDENT_ID          ALPHANUMERIC   (2)
DOMAIN      STU_NAME            CHARACTER      (20)
DOMAIN      YEAR                NUMERIC        (1)
DOMAIN      LOCATION            CHARACTER      (15)

RELATION STUDENT
         (  SID        :  DOMAIN  STUDENT_ID
            SN         :  DOMAIN  STU_NAME
            YR         :  DOMAIN  YEAR
            DISTRICT   :  DOMAIN  LOCATION)
```

Figure 3.3. Domains and Attributes

Relational databases must be composed of relations which meet the following condition:

> Every value in the relation--i.e., each attribute value in each tuple--is atomic (i.e., nondecomposable so far as the system is concerned). [Ref. 4:p. 86]

That is, every data value in the relation is precisely one value. A set of data values is not allowed within the relation. Null values are allowed to represent values in a relation that are not known. When a relation meets the above criteria, it is said to be normalized. This will be discussed in greater detail when the implementation of set-types within a relation is discussed.

The format used to represent a relation is called the relational structure. For example, STUDENT(SID, SN,

YR, DISTRICT) describes the structure of the STUDENT relation. The generic format that describes the relational structure may be said to be Relation_name(attribute1, attribute2,...attributeN). If constraints are imposed on the data values which may exist in the relational structure, then a relational schema exists.

2. Keys

When an attribute with unique values within a relation is used to identify tuples within that relation, that attribute is said to be a key. A person's social security number is an attribute that uniquely identifies that person. Within the STUDENT relation, the attribute SID, is the key. Its unique values distinguish each tuple within the relation.

A key may be the composition of more than one attribute. For example, in the relation ST (Figure 3.1(c)), the combination of the attributes SID and TID serves as the key. A unique tuple within the relation could not be identified if this were not the case. If the key were SID or TID by itself (and not the combination of the two), more than one tuple would have the same key. In the above example, the combination of SID and TID is said to be a composite key. The combination also acts as the primary key for the ST relation. It should be remembered that a relation is a

set, with tuples being elements of the set. Because sets do not contain duplicate elements, each tuple of a given relation is unique in the relation. This property guarantees that the combination of all the attributes will act to uniquely identify a single tuple within the relation.

Relations may exist where more than one attribute combination is able to uniquely identify a tuple within a relation. This relation is said to have more than one candidate key. Our STUDENT relation is such a relation. Tuples (or records) may be uniquely identified within the STUDENT relation by either the value of the SID attribute or by the value of the SN attribute. This is assuming there are never any duplicates in either of the two columns. We may arbitrarily choose either of the candidates to act as the primary key. An alternate key is a candidate key that is not the primary key. In the STUDENT relation SID may act as the primary key and SN would act as the alternate key.

Tuples are uniquely identified by their primary keys depict entities in the real world. Therefore, primary keys serve to uniquely identify entities. Within the STUDENT relation, the tuples represent individual students. The values of the SID attribute identify those students rather than just the tuples that represent

them. As an offshoot of this interpretation, we introduce the following rule.

> INTEGRITY RULE 1 (Entity integrity)
> No component of a primary key value may be null. [Ref. 4:p. 89].

This rule states that all entities must be uniquely identifiable in some way. Primary keys serve as the unique identification function in a relational database. If a primary key were to have a null value in a relation, the entity would be without a unique identification property. This would prohibit that particular entity from being distinguishable from other entities within the relation. It is recommended that both whole and partial null identifiers be prohibited. [Ref. 4:p. 89]

Many times one relation contains references to another relation. Relation ST, for example, includes references to both the STUDENT and TEACHER relations. These references are the attributes SID and TID contained in ST. If a tuple of ST has a value for SID, say S1, then a tuple should exist within the STUDENT relation for student S1. If this were not the case, the tuple containing S1 in the ST relation would refer to a student that does not exist.

To further elucidate matters, we must understand the concept of a primary domain. Date states, "A given domain may optionally be designated as primary if and only if there exists some single-attribute primary key

defined on that domain" [Ref. 4:p. 89]. By adding to its definition (from that shown in Figure 3.3), we are able to specify the domain STUDENT_ID as primary as follows. DOMAIN STUDENT_ID ALPHANUMERIC (2)

PRIMARY

A relation must abide by the following rule if that relation includes an attribute that is defined on the primary domain (for example, relation ST).

INTEGRITY RULE 2 (Referential Integrity)

> Let D be a primary domain, and let R1 be a relation with an attribute A that is defined on D. Then, at any given time, each value of A in R1 must be either (a) null, or (b) equal to V, say where V is the primary key value of some tuple in some relation R2 (R1 and R2 not necessarily distinct) with primary key defined on D. [Ref. 4:p. 89]

The definition of a primary domain implies relation R2 must exist. If attribute A is the primary key of R1, the rule is trivially satisfied.

Finally, we introduce the notion of a foreign key. When an attribute in one relation is the key of another relation, the attribute is called a foreign key. For example, attribute TID of relation ST is a foreign key because its values are values of the primary key of the TEACHER relation.

3. Extensions and Intensions

An extension and intension are components of a relation in a relational database.

The set of tuples existing in the relation at any given instant is known as the extension of that relation. Obviously, the extension is time-dependent. The extension varies as changes are made or operations are performed on tuples (i.e., update, delete, create, etc.).

On the other hand, the intension of a relation is time-independent. The intension directly relates to what is specified in the relational schema. Hence, the intension is the combination of the structure of the relation (the naming structure) and integrity constraints. [Ref. 4:p. 90] The naming structure is the name of the relation and the names of the attributes with their respective domain names. The integrity constraints are divided into key constraints, referential constraints, and other constraints.

The existence of candidate keys implies key constraints. The attribute(s) which make up the primary key and the attribute(s) which make up the alternate keys if any exist, are specified in the intension. A uniqueness restraint is implied by these specifications. Also, the primary key specification infers, in accordance with Integrity Rule #1, a no-nulls constraint. [Ref. 4:p. 91]

The existence of foreign keys implies referential constraints. A specification of all foreign keys in the relation implies a referential constraint. The relations

in Figure 3.1 are examples of extensions. They also show the naming structure (relation and attribute names).

B. RELATIONAL ALGEBRA

Relational algebra defines operations that work on relations. Operators manipulate relations to achieve a desired result. It is very important to note that the result of any operation on a relation creates in another relation. Relational algebra is said to be procedural, which means that the user must not only know what he wants when performing operations on a relation, but also how to get it. As previously mentioned, relations are set with the tuples of each relation considered elements of the set. Therefore any operations that can be performed on sets may be performed on relations which results in a new relation.

Although relational algebra is not often used, it is important to understand as it is the backbone of other high-level query languages such as SQL and QUEL.

The relational algebra may be said to consist of two groups of operators.

1. Traditional Operators

The traditional set operators are union, difference, intersection, and product which are discussed in greater detail below. [Ref. 4:pp. 203-211][Ref. 2:pp. 252-260]

a. Union

Combining the tuples from one relation with those of a second relation to produce a third relation is known as the union of two relations. To make sense, the combined relations must be union compatible. Union compatible means each relation must have the same number of attributes, and the attributes in matching columns must come from the same domain. For example, if one relation (relation A) is the set of students living in Monterey, and another relation (relation B) is the set of students who play tennis, then A UNION B would be the set of students (relation C) who live in Monterey or play tennis (or both). It should be noted that duplicate tuples are eliminated.

b. Difference

A third relation, (relation C) that consists of tuples which occur in relation A but not in relation B is said to be the difference of relations A and B. For example, using the same relations used in the UNION example, A MINUS B is the set of students who live in Monterey and who do not play tennis. Note the relations must be union compatible.

c. Intersection

A third relation (relation C) containing common tuples between two other relations (relations A and B) is said to be the intersection of relations A and

B. For example, again using the relations used in the above examples, A INTERSECT B is the set of students who live in Monterey and play tennis. Again, the relations must be union compatible.

d. Product

The concatenation of every tuple of relation A with every tuple of relation B (resulting in relation C) is known as the Cartesian product of relations A and B. If relation A has m tuples and relation B has n tuples, then their product has m x n tuples. This product is represented by A X B or A TIMES B. For example, the students living in Monterey TIMES a student identification number relation is the set or relation of all possible student living in Monterey/student identification number pairs.

2. Special Relational Operators

The special relational operators are selection, projection, join, and division.

a. Selection

The selection operator outputs a horizontal subset (row) of a given relation. A tuple is now included in the new relation. A tuple may be extracted from a relation by specifying the relation name followed by the keyword WHERE followed by a conditional statement. This conditional statement involving attributes is a single expression or may involve a combination of Boolean

expressions. Figure 3.4(a) shows the selection of the relation STUDENT WHERE SN = "ROSE". Figure 3.4(b) shows the selection of STUDENT WHERE YR > 1. Figure 3.4(c) shows the selection of STUDENT WHERE YR < 3 AND DISTRICT = "SAL".

SID	SN	YR	DISTRICT
S1	ROSE	1	MTY

(a)

SID	SN	YR	DISTRICT
S2	STONE	2	SAL
S3	VENTURA	3	SAL

(b)

SID	SN	YR	DISTRICT
S2	STONE	2	SAL

(c)

Figure 3.4. Selection of Courses Relation

b. Projection

Specified attributes are selected from a relation by the projection operation (projection draws out columns from a relation). The operator outputs a vertical subset (column) of a given relation. Any duplicate tuples within the attributes selected are eliminated. An attribute may only be specified once in a

projection operation. The operator may also be used to change the arrangement of the attributes of a relation because the attributes are specified in a left-to-right order. Figure 3.5 shows the projection of TEACHER on the COURSE and CREDIT attributes. The projection is denoted by TEACHER [CREDIT, COURSE] (relation name followed by attributes to be projected in brackets). Notice that the order of the attributes from the original relation is changed due to the order they appear in the brackets. Duplicates would have been eliminated had they been present.

CREDIT	COURSE
2	HIST
4	MATH
3	ECON
3	PSYCH

Figure 3.5. Projection of Teacher Relation

c. Join

A combination of product, selection, and sometimes projection is known as a join. A JOIN B denotes a join between relations A and B. To join two relations in a natural join, there are three steps. First take the Cartesian product of the two relations, then do a selection to eliminate some tuples. Then if desired, remove duplicate attributes through projection.

If duplicate attributes are not removed it is called an equijoin. When the term join is used, it is understood to mean a natural join. The natural join of the STUDENT and TEACHER relation over the attribute DISTRICT appears in Figure 3.6.

SID	SN	YR	DISTRICT	TID	COURSE	CLSIZE	CREDIT
S1	ROSE	1	MTY	T1	HIST	MED	2
S2	STONE	2	MTY	T4	PSYCH	MED	3
S2	STONE	2	SAL	T2	MATH	LRG	4
S3	VENTURA	3	SAL	T2	MATH	LRG	4

Figure 3.6. Natural Join of Student and Teacher Relations Over District

d. Division

If the domain of a unary relation is also a domain of a binary relation, then we may divide the binary relation by the unary relation. This results in a unary relation consisting of the uncommon domain from the binary relation. The resultant relation contains an attribute value of the uncommon domain if its corresponding entries in the common domain contain all the values of the divisor domain. [Ref. 1:p. 150] Figure 3.7 shows an example of such an operation. The relation TEACH acts as the dividend and relation LOCATION is the divisor. The result is ID (ID = TEACH/DISTRICT). In the figure, the only teacher

identification number (TID) for which there is corresponding values of both SAL and CML in TEACH is T2.

TEACH

TID	DISTRICT
T1	MTY
T2	SAL
T3	CML
T4	MTY
T2	CML

LOCATION

DISTRICT
SAL
CML

ID

TID
T2

Figure 3.7. Division Operation

C. RELATIONAL DATA MANIPULATION LANGUAGES

Data Manipulation Languages (DMLs) are languages through which relational databases may be accessed. There are four main methods for manipulating data in a relational database. These are: relational algebra, relational calculus, transform-oriented languages, and graphic methods [Ref. 2:pp. 252-253].

Relational algebra uses standard set operators to achieve the desired result: a new relation. It is procedural (the user must not only know what he wants but also how to get it) and therefore difficult to use.

Relational calculus is non-procedural (the user only tells the system what he wants, not how to get it). The concept was first proposed by Codd who also presented a language based on this calculus, Data Sublanguage ALPHA.

however, this language was never implemented. [Ref. 4:p. 223]

Transform-oriented languages are non-procedural languages. The most popular of these languages is Structured Query Language (SQL). SQL provides retrieval and update facilities as well as many others. Major SQL-based DBMS products include SQL/DS, System R, and ORACLE. [Ref. 2:p. 437]

QUEL is based on tuple relational calculus and is very similar to SQL. It is non-procedural so the user does not need to concern himself with the underlying physical data structure.

Query-by-example (QBE) is a graphical method to access a database. In this method the user is presented with a picture of the structure of the relation. This is suitable only for terminal use and may not be embedded in a host language.

IV. ADVANTAGES OF RELATIONAL APPROACH

A. INTRODUCTION

As mentioned earlier, relational database management systems are enjoying widespread popularity. According to industry observers, it will be the most prevalent DBMS technology for most data-processing applications. Its understandability, data independence, power and ease of use, and theoretical foundation are the earmarks of relational DBMS. The distinctive features of DBMS are described below.

B. RELATIONAL DBMS ADVANTAGES

1. Ease of Understanding

One of the main motivations for the research work that resulted in the relational model was what Codd termed the "communicability objective". [Ref. 5:p. 110] Achieving this objective means users and programmers alike are able to communicate with one another about the database because they both have a common understanding of the data; the model is structurally simple.

The relational DBMS, using tables with rows and columns which are universally understandable, provides a logical view of data. This means that the database concept is more easily understood to many users as opposed to those who understand the hierarchical or

network based models. Navigation through tables is not necessary because there are no pointers connecting the tables. This is in contrast to the hierarchical and network models where one must maintain a position where one is working when performing most operations. It should be noted that while users may logically view the data in the database as a collection of tables, it is rarely stored as such in physical memory. Thus, the user mentally sees the database as being, in all cases, a collection of tables or files with one record type per table.

2. Data Independence

Data independence is a very important feature of any database system. If an application is data dependent, then the storage structure and accessing strategy cannot be changed without having a pronounced effect on the application. Data independence means applications need not worry about how the data is accessed or physically stored. Codd considered the data independence objective as the most important motivation for the research which spawned the relational model. [Ref. 5:p. 110]

Under the relational model, data is accessed by its value in tables, not by its location. Likewise, tables are related by value, not by pointers connecting

the tables. Also, the system determines access paths, not the program.

Two reasons why applications must be data independent are [Ref. 4:p. 13]:

- Different applications need different views of the same data.
- The DBA must not be forced to change existing applications if he desires to change, due to changing requirements, the storage structure or access strategy (or both).

Data independence greatly affects both end-users and programmers. From the end-users viewpoint, no knowledge of the physical database is required for access; data ordering in tables is insignificant. Therefore, requests for data may be non-procedural. This lack of having to know the underlying data structure combined with the ability to just let the system know what the user wants (and not also how to get it) enables users with little computer/database experience to immediately interact with the system.

From the programmers vantage point, data independence also offers advantages. Many hours are spent making changes to existing application programs. For example, if relational queries are embedded in a host language application program and the nature of the data changes, thus the current path to the data is no longer optimal and the relational query language program only needs to be recompiled. [Ref. 6:p. 93] However, if the

same situation occurred in a CODASYL database, the program would have to be rewritten by someone who understands the effects of the data change on the application program. The time saved due to the concept of data independence may be spent writing new applications for which there tends to be a shortage of.

As organizations grow, there are sure to be desired changes in data representation and growth in data types. Data independence affords organizations this luxury without inflicting damage upon their application programs.

3. Power and Ease of Use

Relational operations are powerful and easy to use because they enable users to operate or process multiple sets of records at a time in a single statement. Hence, relational operations are less procedural. For example, in our TEACHER relation, the SQL statements

```
SELECT   tid
FROM     teacher
WHERE    district = "MTY"
```

would operate on the entire relation (versus a record by record operation as is done conventionally. The advantage of being able to process whole tables of data at one time is obvious. Relations (or sets) are treated as operands in the relational approach.

Less procedurality promotes increased productivity as the retrieval and update operation burdens are placed on the system.

4. Theoretical Foundation

Unlike the hierarchical or network structure, the relational DBMS are based on a theoretical relational model. Relational systems used this model as a foundation while network and hierarchical DBMS have evolved over the years to accomplish as many requirements as possible. Because relational systems have a sound theoretical foundation, the results of relational operations are easily predictable.

The next chapter examines deficiencies of the relational model and offer solutions to these shortcomings.

V. THE RELATIONAL APPROACH: PROBLEMS AND SOLUTIONS

A. INTRODUCTION

Despite growing popularity and the belief that relational DBMS will be the system of choice in the not-too-distant future, they are not without their faults. The performance of relational DBMS is not yet comparable to the standards set by the more familiar models (i.e., network and hierarchical). This performance degradation comes from heavy input/output rather than from pure number-crunching [Ref. 7:p. 26] and is therefore less responsive to technological improvements. Relational systems require significant amounts of CPU and memory resources, and file (database) maintenance is very time consuming. Others claim relational DBMS are not well-suited in a transaction-heavy environment because of lengthy run-times, that the relational concept is unnecessarily complex, and that relational DBMS are very weak in the areas of database recovery and backup. However, to many database specialists and researchers, the main limitation of relational systems is their semantic meagerness. This chapter will focus on the problems caused by this semantic meagerness and will offer solutions to these problems. The main goal will be on easing the formulation of queries and increasing the system's semantics.

B. MENUS AND VIEWS

Database technology was an esoteric subject only a few years ago. It is now of interest to all organizations. Relational database technology, always a favorite of academicians (probably because of its theoretical foundation), is now seeing widespread practical applications. Due to the rapid improvements in and importance of the technology, users with little computer experience and even less database experience must frequently interact with the organization's database.

1. Problems

To a naive seeker of information from the database via a high-level query interface, it is not at all obvious why one question may be asked and another may not be asked [Ref. 8:p. 113]. The naive user expects certain queries to be answered such as:

```
SELECT NAME
WHERE job_title = "manager"
```

but does not understand why a query such as "How is Paul Smith related to the Personnel department?" cannot be answered (assuming of course the user is first able to formulate a query to retrieve such information). That the second query is a query on meta-data and there does not exist a querying facility for database schema (as opposed to data values) does not occur to the user. This lack of familiarity and knowledge of the relations within

the database places a constraint on the user that causes him frustration and inefficiency while using the system.

In many government and commercial databases, relations of a high degree (thirty or more) are not uncommon [Ref. 9:p. 380]. Users should not be burdened with remembering the attributes of each relation (much less even lower-level details such as the format and units of the attributes).

2. Solutions

We suggest the implementation of a command called MENU which allows the user to become familiar with the names of a relation's attributes and its respective domains. The command is typed, followed by the name of the relation with which the user has questions about. For example, if the user types:

```
MENU TEACHER
```

The system would reply:

```
Relation = TEACHER
TID          alphanumeric
COURSE       string
CLSIZE       string
CREDIT       integer
DISTRICT     string
```

This command, at the user's fingertips, provides a quick "explanation" of the relation or a refamiliarization for experienced users. This is more effective than a hardcopy located in a manual which may be lost, mutilated, not updated as the relations change, and which

is time-consuming and difficult to use (i.e., locating the appropriate information).

A reference attribute [Ref. 10:p. 288] has an entity occurrence as its value. If an attribute is a reference attribute (i.e., its domain is another entity or relation) the system must indicate this. For example, if the domain of the DISTRICT attribute in our TEACHER relation was no longer just of type string but was now a relation itself consisting of attributes MAYOR and POPULATION, the system should respond to the command MENU TEACHER with the following :

```
Relation = TEACHER
alphanumeric
COURSE      string
CLSIZE      string
CREDIT      integer
DISTRICT    MAYOR           string
POPULATION      integer
```

If the user was somewhat familiar with the TEACHER relation but was not quite sure of a particular attribute's properties (such as its domain), the system should allow the user to access that particular attribute. A reference attribute and its attributes and their respective domains may be examined by entering a command in the familiar dot-notation. In this example the user may type "MENU TEACHER.DISTRICT." The system would respond with :

```
Attribute = DISTRICT
MAYOR               string
POPULATION          integer
```

The system should allow such notation to any degree of nesting. It is obviously in the database administrator's best interest to limit this degree of nesting to a reasonable level to provide the users with a less complex database.

"The totality of data in a data bank may be viewed as a collection of time-varying relations." [Ref. 9:p. 379] It is due to these changes in the make-up of the relations in the database that a command such as MENU and its extensions must be available.

Another solution to the problem of users' unfamiliarity with relations within the database would be to have a command that when called presents a graphical representation of a particular relation (humans remember pictures very well). This command serves the same purpose as the MENU command mentioned earlier, but may provide a more concrete image of the tabular data structure to the user. The actual data in a relation the user desires to view will not be shown in order to prevent any possible unauthorized access. A skeleton of the relation, with the name of the relation, its attributes, and their domains is displayed. If an attribute is a reference type, the skeleton would name the attribute and its associated attributes and their domains. This relation (or reference type attribute) within a relation should be able to be viewed by the user

through "relation reduction", the technique of examining relations within other relations. A command such as "VIEW TEACHER" would show (see Figure 5.1) the skeleton of the TEACHER relation. If the DISTRICT attribute wants

TEACHER

TID	COURSE	CLSIZE	CREDIT	DISTRICT
ALPHANUMERIC	STRING	STRING	INTEGER	REFERENCE

Figure 5.1. Command View Teacher

to be examined, the command "VIEW TEACHER.DISTRICT" should present to the user the attribute name (in this case, DISTRICT), the name of the owner relation, and its associated attributes and domains (See Figure 5.2). The MENU and VIEW commands are very simple commands which provide users with a fingertip tool to examine closer

(TEACHER)
DISTRICT

MAYOR	POPULATION
STRING	INTEGER

Figure 5.2. Command View Teacher.District

each relation within the database. These commands will enable users to have a better understanding of the semantics of each relation which will aid in query formulation.

C. SET-TYPE ATTRIBUTES

As mentioned previously, a major limitation of the relational model is its semantic meagerness. This meagerness is exacerbated by the lack of a set as a type for an attribute.

1. Problems

The lack of a set-type restricts relational schemas from modeling in a complete and expressive way the real-world relationships between entities which may be any distinguishable thing or object). [Ref. 10:p. 286]

An example will illustrate the problems of not having a set-type attribute with respect to expressiveness shortcomings. Normalized systems require a record to be a collection of single-valued facts. In other words, each data item within a record must be an atomic (nondecomposable) value. A relation where this is the case is said to be in first normal form (1NF). A more complicated data structure than a two dimensional table with homogeneous entries in the columns would be needed if the 1NF requirement was not met. Figure 5.3 is a relation we will call SCHOOL. If the relation wishes to depict the instructors each particular student has then a tuple must exist in the relation for each individual instructor that student has. This is called a nomalized relation. However, Figure 5.4 condenses this relation by relaxing the 1NF constraint. Whereas the

SCHOOL

SID	YR	INSTRUCTOR
126	SOPH	SMITH
126	SOPH	JONES
126	SOPH	DRAKE

Figure 5.3. Relation School

SCHOOL

SID	YR	INSTRUCTOR
126	SOPH	SMITH, JONES, DRAKE

Figure 5.4. Relation School with Set-Type Attribute

schema previously would have been SCHOOL (sid : INTEGER; yr : STRING; instructor : STRING) with a set-type implemented it becomes SCHOOL (sid : INTEGER; yr : STRING; instructor : SET_OF STRING). The attribute INSTRUCTOR is now a set-type attribute. The set-type saves space by reducing redundancy and truly allows the record to contain all the facts about an entity [Ref. 8:p. 115]. By retrieving this one record, the user will see the natural or true relationship between the student and his instructors. If the INSTRUCTOR attribute was not a set-type, a record that was retrieved would not tell the whole story (i.e., the user may believe the instructor retrieved from one particular record was the students' only instructor which may not be the case). A

lack of a set-type in addition to placing a restriction on a record from possibly containing all the facts about an entity creates other problems. Many real-world entities do not have single-valued information which can easily be represented within the field of a record [Ref. 8:p. 116]. For example, a relation called TEAM may not happen to have any single-valued facts. The team may have numerous players, many games, several coaches, and a large number of potential recruits. A number of relations (i.e., TEAM_PLAYERS, TEAM_COACHES, etc.) each containing an attribute is necessary to store the data rather than simply one relation called TEAM.

Many changes have been proposed to enable the relational model to better capture the semantics between real-world entities. Some have proposed adding semantic data model capabilities to relational systems thus combining the advantages of both models [Ref. 10:p. 286]. Others have proposed extensions to the relational model, such as Codd's extended relational model RM/T.

However, relaxation of first normal form (1NF) constraints is the most common proposal. This proposal, in effect, would allow attributes to be of a set-type. As mentioned earlier, the lack of a set-type inhibits the relational model's expressivity. Another problem which arises from the lack of a set-type attribute is query formulation. Asserting that the lack of a set-type is a

main cause of user difficulties in formulating many queries, focus will now be directed to this problem.

To make the query formulation issue more transparent, we introduce four relations. The key of the relation is the attribute or set of attributes underlined. [Ref. 11:p. 2]

student(sname, district)
studies(sname, course)
developed(tname, course)
teacher(tname, district)

As stated earlier, due to the lack of a set-type some entities are inappropriate for storage in a single relation. The TEAM relation served as an example of this. With no single-valued attributes, a number of smaller degree relations would better represent the entity.

For the same reason, in the above example we are forced to create a separate relation, STUDIES, which lists the courses a particular student is currently studying. If a set-type been implemented, this data could have been stored in the STUDENT relation, giving the user a more concise, semantic view of the data.

Why does the lack of a set-type make user query formulation a more difficult process? In the above example, two relations are needed (STUDENT and STUDIES) where if a set-type were available, one would suffice. When a user attempts to formulate a query, he must go through the mental gymnastics of joining relations to

result in the desired final relation from which the data he desires will be retrieved. This is not an easy task for users of any experience level. Recent criticism of relational systems has been the difficulty users have in understanding joins. Languages such as SQL and QUEL are manageable for programming but not for users; and the reason is brought to light with joins [Ref. 12:p. 70].

An example best demonstrates the query formulation problem. Consider the following query in SQL form:

```
SELECT student.sname, teacher.tname
FROM   student, teacher
WHERE  student.district = teacher.district
```

This is obviously a very simple query. Only two relations are required to be joined. The equivalent relational algebra expression is :

```
PROJECT sname, tname (student |X| teacher)
```

The projection retrieves the (sname, tname) pair who live in the same district.

For an experienced programmer or user, the translation of "Find the names of the students and teachers who live in the same district" into a query is not very difficult. For an inexperienced user, the mental manipulation required to answer this query is not quite so easy but yet still manageable. If sname and tname been set-type attributes, the TEACHER and STUDENT

relations could have been combined into one relation in the beginning (a new relation called STUD_TEACH) and the query would have been reduced to:

```
SELECT  sname, tname
FROM    stud_teach
WHERE   stud_teach.district = X.
```

As can be seen, the gain (query-wise) of having set-type attributes in this particular case is minimal. This will not be the case with more difficult queries.

Let's consider a more challenging query:

```
SELECT sname
FROM   student
WHERE  district  IN
    (SELECT district
    FROM   teacher
    WHERE  tname  IN
        (SELECT tname
         FROM   developed
         WHERE  course = "math")).
```

Translating this query into its English equivalent is not so easy. However, this cannot be attributed to SQL syntax, which in this case, is very basic. Since any query one can formulate in SQL may also be formulated in QUEL, write the query in QUEL-like structure to see if this contributes to the semantic of the query :

```
SELECT student.sname
FROM   student, teacher, developed
WHERE  student.district = teacher.district  AND
        developed.tname = teacher.tname      AND
        developed.course = "math".
```

This is equivalent to:

```
PROJECT sname
(PROJECT sname, tname (student |X| teacher)  |X|
PROJECT tname (developed WHERE course = "math")).
```

The first nested projection retrieves the (sname, tname) pair where sname and tname live in the same district. The second nested projection retrieves tname after it has been selected where tname developed the math course. These two relations are joined over the attribute tname, and sname is projected from this relation.

This transformation to a QUEL-like query does little to help us with the semantics of the query. This is because one still must visualize the query as a series of natural joins occurring between the relations over common attributes. As earlier expressed, humans do not easily digest joins, especially more than a couple taking place within the same query. The English translation, "List the students who live in the same district as the teacher who developed the math course" is difficult to formulate into a query because it requires joins of three separate relations.

One more example will further illuminate matters:

```
SELECT  sname
FROM    student
WHERE   district  IN
    (SELECT district
    FROM   teacher
    WHERE  tname  IN
        (SELECT tname
        FROM   developed
        WHERE  course  IN
            (SELECT course
            FROM   studies
            WHERE  sname = student.sname))).
```

This query is still more difficult. Again, this is not due to the SQL syntax, but may be attributed to another situation where multiple joining of relations is required. The equivalent query in "QUEL-like" SQL structure is:

```
SELECT student.sname
FROM   student, studies, developed, teacher
WHERE  student.district = teacher.district  AND
       student.sname = studies.sname        AND
       studies.course = developed.course    AND
       developed.tname = teacher.tname
```

This is equivalent to the following relational algebra expression:

```
PROJECT  sname  (PROJECT  sname,  tname  (student |X|
teacher)
      INTERSECT
            PROJECT  sname,   tname   (developed  |X|
            studies).
```

Here, the first nested projection retrieves the sname, tname pair where sname and tname live in the same city. The second projection retrieves the same pair where sname studies the course developed by tname. The intersection and subsequent projection results in a list

of the student names who live in the same district as the teachers who developed the courses they are studying.

This query in relational algebra is semantically more clear to users as a set operator, namely intersection, appears in the expression. The difficult task of joining relations is "softened" by the presence of the intersect operator which serves to re-establish the set concept in the user's mind. The concept of natural joins, much less a sequence of them, is not supported or strengthened by real-world occurrences with which users may identify. On the other hand, users naturally compartmentalize and willingly think in terms of sets but do not often mentally perform natural joins.) Thus the lack of a set-type, by requiring multiple joins to take place, forces users to think in terms of a physical database structure and a concept they cannot relate to the real world (joins) as opposed to the natural concept of sets and set operations.

The notion that each relation within the database is a set where each tuple (or record) within the relation is an element of that set is inconsistent within the user's mind. When the user employs basic set operators such as SELECT, he expects a list or set of X to be retrieved from the relation specified. When a join (or multiple join) is performed, the user must mentally sequence through a series of steps that deal with lower-

level concepts. This "low-level traversal" causes the user to lay aside the thought of the set to be retrieved until the notions of common attributes, domains, etc. are dealt with.

2. Solutions

The proposed solution to the problem involves the obvious addition of a set-type to relational DBMS. Any attribute may be declared of type set within the database schema. The declaration is of the form : SET_OF (type or entity name). Our sample database schema may be defined as follows:

```
student (name : STRING; instructor    : SET_OF develop;
district :
         STRING)
develop (name : STRING; course : STRING; classes : SET_OF
study)
study (name : STRING; course : STRING)
teacher (name : STRING; district : STRING) [Ref. 11:p. 7]
```

The query "List the students who live in the same district as the teacher who developed the math course" may be programmed as follows:

```
    SELECT student
  WHERE  student.instructor  CONTAINS
            develop WHERE
               develop.course = "math"
     AND    student.district = teacher.district
```

Noticeably absent from this query is the presence of a join. Although more than one relation is involved in answering the query (student and develop) the user does not see the explicit join. Instead, from the student relation we "placidly migrate" into the develop

relation. This more delicate process is easier for users to see mentally.

The query "List the students who live in the same district as the teachers who developed the course they study" may be expressed as follows:

```
    SELECT student
  WHERE  student.instructor  CONTAINS
            develop WHERE
               develop.classes  CONTAINS
                  study WHERE
                     study.course = course
  AND     student.district = teacher.district
```

Again, this query, when compared to the previous query (without set-types), is easier for users to understand and to formulate due to the removal of explicit joins. The user is again immune from having to think about lower-level items such as attributes and common domains for joining purposes.

It should be noted that the DBMS with the set-type attribute implemented took significantly fewer physically coded lines to achieve the same result as the system without the implementation of a set-type. "List the students who live in the same district as the teachers who developed the course they study" required twelve lines to formulate versus seven with the implementation of a set-type. Likewise, "List the students who live in the same district as the teacher who developed the math course" required nine lines of code without a set-type implemented and five lines of code

with the set-type implemented. This is directly credited to the set-type attributes which eliminate the need for the nested SELECT operation. The SELECT operation is what ties together attributes in a complex nested query. However, this operation is made obsolete in the set-type attribute example since the nested queries within a complex query do not call for one specific attribute value but require the value to be a member of a set of relations.

Also, the query with the set-type attribute implemented is not so disjointed with respect to readability. The query reads more easily than the non-set-type attribute query which is very choppy and mentally difficult to tie together. Although the degree of nesting is the same in the two queries, the set-type query is more concise and manageable. This conciseness is due to the nested queries within the whole query being sewn together by common relations. While the conventional query is tied together by common attributes (thereby requiring the relations to be explicitly named within each nested query with a FROM clause), the set-type query is tied together by sets of relations. Conciseness is important as "query recognition" is also an issue that must be addressed.

The bedfellow of "query formulation" may be considered "query recognition". Query recognition is an

important issue for a variety of reasons. If faulty or unexpected data is retrieved from the database, the query must be examined to see why this occurred. If it cannot be determined what the query is trying to accomplish then a real problem exists. A serious deficiency in the make-up or design of the database may exist which goes unnoticed. Obviously, the more time and resources expended to determine the purpose of the query (i.e., what the query is doing or trying to achieve), the more costly the process is to the organization. This assumes that the answer to the query is "caught" or determined by the user to be wrong. The ability to troubleshoot queries is much easier if a set-type attribute is implemented in the system. As mentioned earlier, the readability issue coupled with the common-relation thread factor makes this possible.

Query recognition is also important since new queries that must be formulated may be based on or structured around other queries which have already been formulated. The chance of a user formulating a correct query on the first try is very slim. If a template or similar query is available to follow, then the task is made much easier. A user is more apt to recognize a query with a set-type attribute implemented because there are fewer physical lines required to evaluate or "debug". The trick here is finding that template or

similar query. The set-type implementation lends itself well to this purpose due to readability coupled with fewer lines to evaluate.

To this point we have discussed implementation of set-type attributes in our relational schema (i.e., instructor: SET_OF develop and classes : SET_OF study) with respect to query formulation. Now examine the implementation of a set-type with respect to system efficiency will be examined. A relation called TITLES (see Figure 5.5) consists of three attributes : NAME, YRS_PRO, and YR_TITLE_WON. The domains of the attributes are STRING, INTEGER, and INTEGER respectively. Again, without the implementation of a set-type, notice

TITLES

NAME	YRS-PRO	YR_TITLE_WON
JONES	5	1974
JONES	5	1976
JONES	5	1978
SMITH	4	1982
SMITH	4	1984
SMITH	4	1986
STEED	6	1974
STEED	6	1976

Figure 5.5. Relation Titles

the redundancy required to represent the data. Note also the each record only tells part of the story as each professional in this example, won a title in more than just the year indicated in a particular record. Now formulate a simple query that says, "Name the years, if any, in which Jones has won tournament titles." The query in SQL form would be:

```
SELECT    yr_title_won
FROM      titles
WHERE       name = "Jones"
```

To retrieve the proper answer to the query, the system had to find the first record where the value of the name attribute was "Jones" and then retrieve the data value of the YR_TITLE_WON attribute. In this particular relation, there are three records with the name "Jones" so all three records are retrieved with respect to the YR_TITLE_WON attribute value.

Consider another relation called TITLES2 (see Figure 5.6) which consists of the same attributes as the relation TITLES, specifically NAME, YRS_PRO, and YR_TITLE_WON. The domain of the NAME and YRS_PRO attributes remain the same (STRING and INTEGER respectively). However, the YR_TITLE_WON attribute in TITLES2 is a set-type attribute. Its domain is now SET_OF INTEGER. This is similar to the "instructor" and "classes" attributes in our previous schema, the only difference being their domains were sets of other

relations. To extract all the years "Jones" won titles then it can be done with the following query:

```
SELECT   yr_title_won
FROM     titles2
WHERE    name = "Jones"
```

This assumes that a high_level query language is able to retrieve all the elements of a set-type attribute which will be discussed in greater detail later.

TITLES2

NAME	YRS_PRO	YR_TITLE-WON
JONES	5	1974, 1976, 1978
SMITH	4	1982, 1984, 1986
STEED	6	1974, 1976

Figure 5.6. Relation Titles2

Although the query is, except the name of the relation exactly like the previous query, it only requires one record to be accessed. The set-type attribute saves the system from having to search for other records with the name "Jones".

Other queries may also have an effect on system efficiency. For example, if the user wishes to know the total number of titles "Jones" has won, the system must search the relation for each entry with the name "Jones" and keep a running total of such entries. This assumes that there is an entry only if a title was won that

year. With the set-type attribute implemented, a query could be constructed as:

```
SELECT   count(yr_title_won)
FROM     titles2
WHERE    name = "Jones"
```

An assumption is made that the atomic items within the set of a set-type attribute are distinguishable. This query is similar to the query with the system without a set-type implementation. Such a query would be:

```
SELECT   count(*)
FROM     titles
WHERE    name = "Jones"
```

The difference lies in the work the DBMS must do. The system "remains" at the appropriate record with the set-type attribute while the count operator totals the titles won. Without a set-type attribute, the system must again sequence through the records.

Had the domain of the YR_TITLE_WON attribute included a null value (i.e., the relation included other attributes therefore a record in the relation no longer implied a tournament title), the system would have to have done more work checking to see if in fact a title was won. Each YR_TITLE_WON attribute value would had to have been specifically checked (i.e., the presence of a record with the value "Jones" would not be enough to assume a title win.)

A set-type attribute which is a user-declared type adds flexibility to relations and query formulations. For example, define a type as follows:

```
type AFTER_YR = integer > 1976.
```

Our schema for relation TITLES2 is :

```
titles2(name     :    STRING;   yrs_pro    :    INTEGER;
yr_title_won :
          SET_OF INTEGER).
```

Now suppose the organization utilizing the database is only interested in titles won after 1976. The domain of YR_TITLE_WON without a set-type implemented should now be integers greater than 1976). A domain change is desired so faulty updates will not occur and improves system performance since records will now be eliminated from the relation that do not contain applicable values. The relation would have to be modified as the tuples with values less than 1977 under the YR_TITLE_WON attribute are no longer valid. This is harmful for two reasons. Firstly, the data that will exist in the relation is no longer the complete data for an individual player. Tournament victories prior to 1977 are no longer represented therefore the relation does not describe the real world accurately. Secondly, if the organization wishes to have this data available at a later time the relation must be modified again.

The set-type attribute combined with the previously user-declared type AFTER_YR precludes changing

the relation TITLES2 which does have YR_TITLE_WON attribute values less than 1977. For example, if the years that "Jones" won titles after 1976 were selected, the query could be formulated as:

```
SELECT   yr_title_won
FROM     titles2
WHERE    name = "Jones"    AND
         yr_title_won      IN   after_yr.
```

The query retrieves the desired data while the relation, which did not have to be modified, accurately represents the real world.

Because the user is not required in the query to explicitly state the year (i.e., 1976) after which "Jones" won titles, greater flexibility is achieved. Should the organization decide to change or impose different constraints on the data to be retrieved from the database, the DBA must only change the value of the user-declared type. For example, if at a later time the organization is only interested in titles won by players after 1980, we simply change the user-declared type to "type AFTER_YR = integer > 1980." The user need not necessarily be aware of this change. All the user must know is how to formulate the query which is the exact process used previously. As long as the user is aware of the names of the user-declared types, he may follow the template of previous queries. A standard set of user-declared types could be implemented in queries by users with only the DBA having to know the underlying

details. For example, an organization may be interested in titles won before, during, and after a certain value which changes on a regular basis. Types could be declared (i.e., BEFORE_YR, DURING, and AFTER_YR) and users could use in their queries. The regularly changing value or reference is of no concern to the users. Aside from the added flexibility, time and resources are saved since no changes to the database are necessary.

D. QUERY LANGUAGE MODIFICATIONS

The implementation of set-type attributes in a relational schema requires examining the implications of such an implementation to query languages. The first normal form (1NF) requirements cannot be relaxed without considering the effect such a relaxation will have on our ability to retrieve, update, and store information in the database.

As mentioned earlier, the 1NF requirement ensures all data within the relation will be nondecomposable (that is, atomic in nature). This means that each piece of data retrieved will be atomic or single-valued. Likewise, the database may only be updated with similar values such as those from the same domain.

1. Problems

The need to address the issue can be clearly seen by retrieving the INSTRUCTOR attribute values from the

SCHOOL relation (see Figure 5.3). "Name an instructor of student 126" would normally be programmed as follows:

```
SELECT    instructor
FROM      school
WHERE     sid = "126"
```

However, if the attribute (in this case, INSTRUCTOR) is a set-type, all the data is retrieved. This presents problems if there are numerous values within the attribute. In this instance, all three values (SMITH, JONES and DRAKE) would be returned as the answer to the query.

A user-invisible relation is the most common proposal for storage of set-type attributes. For example, the values of the INSTRUCTOR attribute in the SCHOOL relation would be stored in a separate relation as shown in Figure 5.7. Naturally, a reference must exist

VALUE	INSTRUCTOR
1	DRAKE
4	JONES
7	SMITH

Figure 5.7. User Invisible Relation

which connects this separate, user-invisible relation to the tuple which "owns" it.

Common proposals have been for the adoption of the familiar dot notation to retrieve data from attributes which are reference attributes. It is

considered that a reference attribute to be an attribute which has an entity occurrence as its value [Ref. 10:p. 288]. For example, the attribute CLASSES in the following schema would be considered a reference attribute:

```
develop(name : STRING; classes : STUDY)
study(name : STRING; course : STRING).
```

Thus, as expected, "develop.name" is a string while "develop.classes" is an entity occurrence of type STUDY. To retrieve a course name the following notation must be used:

```
SELECT  develop.classes.course
WHERE   develop.name = "Smith".
```

This notation eliminates the need for the user to develop a query with a complex join statement. As previously mentioned, a join is not generally an easy concept for users to understand. If the dot notation had not been implemented, the query would have been:

```
SELECT    course
FROM      develop, study
WHERE     name = "Smith".
```

The dot notation query formulation is more concise and easier to formulate for two reasons: First, an in-depth knowledge of a relations attributes, specifically their semantics, is not needed since a join is not explicitly performed. Although joining over the NAME attribute from the DEVELOP and STUDY relations is in this example semantically correct, there is no guarantee

of semantic correctness in all cases. The dot notation prevents users from having to deal with low-level concepts such as attribute names and allows them to maintain the mental model of a set while formulating the query.

2. Solutions

The dot notation that is applied to reference attributes may be modified to handle set-type attributes. It is suggested that the following implementation enables users to retrieve from a set-type attribute a specific value.

Each atomic value within the set-type attribute (such as SMITH or JONES) should be stored within the user-invisible relation with a numerical value which uniquely identifies that value within the relation (see Figure 5.7). Notice that the values within the relation are ordered. If the user desires to know all the values of the set-type attribute, then the query is formulated in the usual manner. However, if a certain or individual value is desired to be retrieved, then the query language must be modified so the user is able to retrieve that value. If a query such as "Name an instructor of student 126" is to be formulated, the query language should be modified as follows:

```
SELECT   instructor.1
FROM     school
WHERE    sid = "126".
```

This notifies the system that the first value or element within the attribute is the item of interest. Note that the "1" in the query has no user-known link to the "1" stored next to the value "Drake" within the user-invisible relation that stores the instructor data and their corresponding integer values. The "1" in the query merely designates the first value within the set-type attribute. To retrieve the next value, the SELECT statement in the query would be "SELECT instructor.2" or "SELECT instructor.>" to signify an incremental increase of one with respect to the location of the value within the set-type attribute. Thus, either of the two SELECT statements above, together with the FROM and WHERE statements, would retrieve the value of "Jones". After the value "Jones" is retrieved, another query with a SELECT statement such as "SELECT instructor.>" would result in the value "Smith", while a query with a SELECT statement such as "SELECT instructor.<" would again retrieve the value of "Drake". In this way we may traverse our way through the set-type attribute. The system must keep track of the last value from the set-type that was retrieved.

The integer values stored with the instructor string values serve as a reference for the system to keep track of the relative positions of the values within the set. When the user types an attribute name followed by

">", it signals to the system to locate the value with the next highest corresponding integer value relative to the last integer value retrieved. This should not be too costly if the items in the user-invisible relation are ordered.

E. JOINS

Joins were discussed in an earlier chapter therefore it is assumed the reader understands the basic concept of joins.

1. Problems

A criticism of relational systems has been that joins are very time-consuming and expensive. Improved techniques in the area of query optimization and indexing have been developed which have assuaged some of the opponents of relational systems. However, many opponents stand by the claim that regardless of the fact that optimization techniques have made joins less expensive to the system and its users, it is still an operation that is a thorn in the system's side due to the difficulty users have in understanding joins. Furthermore, joins that semantically make no sense are still allowed by some relational DBMS products.

The join function, to make sense, must be executed on columns containing field values that are drawn from the same domain. However, this constraint is not enough. A join made over two attributes with common

domains such as prices in one table and weights in another table is allowed but is not in keeping with the spirit of joins much less the semantics. An example will best illustrate the potential problem. Suppose a user wanted to do the following, "List the districts where the employees have a credit of 3 dollars." The query in SQL form would be:

```
SELECT  district
FROM    teacher, employee
WHERE   employee.credit = 3.
```

This query would cause the TEACHER and EMPLOYEE relations (Figure 5.8) to be joined over the common attribute CREDIT. The answers to the query would be CML (Carmel)

TEACHER

TID	COURSE	CLSIZE	CREDIT	DISTRICT
T1	HIST	MED	2	MTY
T2	MATH	LRG	4	SAL
T3	ECON	SML	3	CML
T4	PSYCH	MED	3	MTY

EMPLOYEE

EID	OWE	PAID	CREDIT
E1	50	53	3
E2	30	32	2

Figure 5.8. Relations Teachers and Employee

and MTY (Monterey). Although this may be true, there is no way to guarantee it. The user instructed the system what to do via the query. Being a procedural model, the system performs the join how it sees most efficient. The system is unable to semantically differentiate the two CREDIT attributes, therefore performs the join over the two different attributes, resulting in possible erroneous data. Such a join may be intended but is most likely not. There is no guarantee that a user will always recognize this mistake in logic.

2. Solutions

The following solutions to the problem are suggested. Users must have a basic understanding of a join and how it works. It is not enough that they understand SQL alone. A user may be very well-versed with respect to SQL syntax issues but this does not prevent semantically inappropriate queries from end-users. Although a main benefit of the relational system is that users do not have to understand the underlying structure of the data to use the system, a join is really a simple operation that even naive users should understand. The MENU command mentioned earlier may be expanded to supply the user with not only the domain of each attribute, but also a brief narrative comment about the domain. For example, in response to a user's command of MENU TEACHER the system's reply would be:

```
Relation = TEACHER                  Join traits
TID        alphanumeric        teacher identification #
COURSE     string              course teaches
CLSIZE     string              size of class
CREDIT     integer             course credit hours
DISTRICT   string              city where taught
```

This option will ensure that the user understands that the CREDIT attribute contained in the TEACHER relation would semantically be inappropriately joined with a CREDIT attribute containing an employee's credit balance in an EMPLOYEE relation. The system will do only what it is told to do by the users. This resource may prevent the user from issuing a query, which the system would gladly accept, that may be very harmful to himself by retrieving inaccurate data.

To this point we have burdened the user with making sure the joins have semantically been sound. A more active role by the DBMS is the most effective way to achieve the desired result. An interactive dialogue with the user would prevent some inappropriate joins from occurring. For example, a join between the TEACHER and EMPLOYEE relation would result in the system response:

```
Joins relations :  TEACHER and EMPLOYEE
Join attribute  :  CREDIT
Join traits     :  TEACHER   course credit hours
                   EMPLOYEE  employee's credit balance

        Proceed with Join (Y/N)? >
```

A user with any experience in formulating queries should recognize that such a join is not semantically correct. To further help the user, the system should respond to a

negative answer at the prompt with other possible relations, if any exist, that will provide the user with a possibly semantically correct join. For example, if the attribute CREDIT appears on another relation within the database the system should inform the user of this in such a manner:

CREDIT appears on 3 files: TEACHER
EMPLOYEE
BANK

CHOOSE FILES DESIRED:

Here it is assumed that there is yet a third relation within the database with the attribute CREDIT (namely BANK).

Although the system dialogue with the user does take time, the benefit of a correct answer to his query outweighs this added time. It is assumed that a very experienced user with the database would not need to employ this feature. However, if a query receives an unexpected answer, any user will be able to troubleshoot the cause of the suspected problem.

F. TYPE CHECKING

The real source of the join problem is the lack of strong data type checking.

1. Problems

Database designers are unable to declare their own data types other than system-defined data types. This leads to the strong possibility of semantically

incorrect joins. The lack of strong data type checking may also cause a loss of data integrity within the database. Examples will best illustrate this shortcoming. In the STUDENT relation, the domain of values for the YR attribute is the integers 1, 2, 3 and 4. In the TEACHER relation, the domain of values for the CREDIT attribute is also the integers 1, 2, 3 and 4. Although the two attributes have the same domain, joining the two relations over those two attributes would be semantically incorrect (years in school is hardly equal to course credits).

Without a user-declared data type, constraints on updates are not as easily enforceable and lack of data integrity results. As an example, assume a relation called FLIGHTS exists in our database .

FLIGHTS (e)

ORG	DEST	CAPACITY	TIX#	
MTY	CHI	200	273	EAST ODD —
CHI	LA	151	1242	WEST EVEN —

Figure 5.9. Relation Flights

Also assume that a travel agency, by convention, assigns odd ticket numbers to those passengers travelling East and even ticket numbers to those travelling West. The domain of the TIX# attribute in the FLIGHTS relation is currently positive integers. Due to weak typing, an

agent may assign a wrong ticket number may be assigned to a passenger thus compromising the integrity of the database.

2. Solutions

The implementation of a command by which users when creating a new relation, may declare their own data types is suggested. For example, a new table is usually created using a command such as BUILD in the following manner:

```
BUILD FLIGHTS(org=5c, dest=5c, capacity=1i, tix#=1i).
```

A relation called FLIGHTS is created with four attributes: org (origin)-- with five or less characters, dest (destination)-- with five or less characters, capacity--with a 1-byte integer, and tix# (ticket number), with a 1-byte integer. With a new command called TYPE, the user creating the new relation may declare a set type in such a manner:

```
BUILD FLIGHTS(org : 5c; dest : 5c ; capacity : 1i
              pass# : direction)
TYPE direction = even(integer) if ;travelling west
                 eastof(ord,dest)
                 else odd(integer);travelling east
```

The declaration of type-direction assumes a predicate "eastof" has been defined. When the statement "eastof(org,dest)" is executed, the system must check the value of "org" against the value of "dest". A database of facts concerning the locations of all possible origins and destinations is assumed to be searched by the system

when a new record is inserted in the FLIGHTS relation or when a record is updated. A specific example such as the preceding one serves to show how challenging it is to implement a command such as TYPE into a system. The template of a type declaration is :

```
TYPE typename = code using a high-level language
                with standard operators within
                that language
```

Not all TYPE declarations are so challenging. For example, if a current domain was currently over positive integers and the designer or DBA wanted to restrict the domain to integers greater than eighteen, the declaration would be:

```
BUILD YRS(name : 20c; state : 3c; age : eighteen)
TYPE eighteen = integer >= 18.
```

If a user of the system attempts to update or enter a value within the age field of a record that is less than eighteen, the system should provide a signal of this illegal action. For example, the system may reply:

```
USER ENTRY FOR AGE OUT OF LIMITS
For help type "MENU YRS" or "EXPLAIN AGE"
```

If the command MENU YRS is entered, the system would respond with:

```
Relation = YRS
NAME       string
STATE      string
AGE        eighteen
```

The user now sees that the domain of the AGE attribute is a type called eighteen. A more direct route to the

problem would be to furnish the user with a command that describes user-declared types. We suggest the command "EXPLAIN typename" to be implemented for this purpose. The user enters "EXPLAIN AGE" to which the system replies "TYPE EIGHTEEN :value entered must be greater than or equal to eighteen." The user is now aware of why the previous record was unable to be entered or updated and may take corrective action.

It is obvious that the option of user-declared data types is costly to users with respect to time. The system has much more work to do because of the added constraints imposed by the user. The user benefits from the stronger data typing by preventing a compromise of the integrity of the database. Most would agree that accuracy of the information upon which the organization and/or people base their decisions is more valuable than having not so accurate information a little sooner. There continues to be considerable research effort in the area of doing as much type-checking as possible with as little run time cost as possible. [Ref. 13:p. 43] Until research in this area proves fruitful, the implementation of the system-user dialogue coupled with the implementation of the previously defined commands is suggested.

G. ACCESS PATHS

Access paths must be mentioned when critiqueing relational database systems. They play a major role in the joining of relations and also greatly affect the overall performance of the system. An access path involves the order in which records are read. An access path also involves whether or not indexes are used and the decision of whether to read a record from file one and compare the values with file two, or to first read a record from file two and then compare the values with file one. For example, if a user wanted to find all the students who take the math course (assuming all students take every course taught in their district) then the STUDENT and TEACHER relations must be joined over the common attribute DISTRICT. Depending upon the access path, a record would first be read from the STUDENT relation and then compared with each record in the TEACHER relation or a record would first be read from the TEACHER relation and then compared with the records in the STUDENT relation. In this particular example, because a selection is to be made (the "math" value from the COURSE attribute), it would be wise to make the selection first. This would reduce the number of record instances to be considered for the join. It should be remembered that tables are related by value as opposed to

pointers and that access paths are determined by the system and not by the programmer.

1. Problems

In hierarchical and network systems, access paths are predefined in the data structure. This is taken care of by the DBA. The relational systems differs in that there are no predefined paths in the data structure as seen by the user. Many different paths may exist because access is accomplished by the matching of field values. There are pluses and minuses with these differences. A hierarchical or network diagram, where access paths are predefined and explicitly shown, will in a quick glance provide an immediate understanding of many complex interrelationships. [Ref. 14:p. 37] On the other hand, irony exists as the relational software, which works to buffer the user from access considerations, prohibits users from fully and quickly digesting how the tables are interrelated [Ref. 15:p. 48]. The gains made in the system by reduced procedurality may be lessened as there are no predefined access paths of which the user may take advantage .

We have previously mentioned how time-consuming and inefficient joins may be. In fact, when the relational model was introduced by Codd, it was thought that the inability to choose efficient access paths when answering queries involving joins on random collections

of files would keep the relational system from ever becoming very practical. [Ref. 6:p. 92] Optimization techniques have alleviated the problem. For instance, the System R (IBM's prototype for SQL/DS and DB2) [Ref. 6:p. 94] query processor evaluates different possibilities for using or not using indexes and for making joins in one sequence or another sequence. The designers state that the path that really is least expensive to the system is chosen most of the time.

2. Solutions

The following procedures are suggested for implementation in relational database management systems with respect to joins and access paths.

Attributes over which the relations are joined may be from the same domain but are semantically worlds apart. The database designer or DBA should set up a file which lists relations and the attribute name they have in common. The pair of relations should only be included in the list if joining over the like attribute would be semantically inappropriate. For example, relations EMPLOYEE and TEACHER would be included in the file with the common attribute of CREDIT separating the two (see Figure 5.10). Any time a user attempts a join, the indexed file is searched to see if the join should be allowed. If the system determines the join is allowed,

the system proceeds with the join. If the two relations and the joining attribute is in the file, the system

R1	COMMON ATTRIBUTE	R2
EMPLOYEE	CREDIT	TEACHER

Figure 5.10. Relations with A Common Attribute.

should alert the user. For example, "JOIN NOT ALLOWED. SEMANTICALLY INCORRECT." There should be an option for the user to find out the other relations which contain the common attribute. A command such as "FIND attribute" would suffice. For example, the command "FIND CREDIT" would result in the system responding:

```
TEACHER     course credit hours
EMPLOYEE    credit balance
```

The file is, in effect, a Boolean type operator which determines whether or not the relations may be joined. The database designer or DBA must be careful when new relations are added to the database. Furthermore, each operation on a relation creates a new relation that must be taken into account. An algorithm must exist that places the attribute already contained in a relation in the file along with the other relations that contain the attribute.

Along with the prevention of improper joins, the system should notify the user when there is no way to

make a join. A system response to a join unable to be made may be:

> NO JOIN ABLE TO BE MADE. TYPE "FIND attribute_name" FOR OTHER POSSIBLE RELATIONS WITH DESIRED JOINING ATTRIBUTE

If there exists more than one way to make joins, the answer should be arrived at by the system via the shortest path. This is an issue that the database designer must deal with. However, if the designer is unable to avoid a situation whereby there is more than one way to answer the same query, the user should be aware of this. As mentioned earlier, a user-dialogue where the system provides the user with the chance to pick the relations over which the join will be performed should be available. For example, a response to a query that may be answered by the joining of different combinations of relations may be:

> RELATIONS CHOSEN FOR JOIN ARE : TEACHER AND EMPLOYEE
> ARE YOU SATISFIED WITH THE RELATIONS CHOSEN? (Y/N) >

An affirmative answer would instruct the system to proceed with the join, a negative answer should instruct the system to provide the user with the other relation(s) that may be chosen for joining.

Relations which are semantically joined improperly is not the only source of problems with joins. A user without a security clearance must not be able to retrieve sensitive data from the database if he formulates a query which causes a relation he has

authorization to access to be joined with a relation which contains sensitive data.

Whether or not the relation is sorted and the presence (or absence) of indexes on the join attributes also plays a major role in the performance and efficiency of joins. The importance of access paths, which so greatly affect the performance of joins, and hence the relational DBMS performance cannot be understated. It is for this reason that the previous recommendations place a heavy emphasis upon the semantics of any joins performed.

VI. CONCLUSIONS

Database processing technology is a rapidly growing field with the future promising continued growth. Organizations are becoming increasingly dependent upon their database processing capabilities in order to remain competitive. Organizations which do not effectively and efficiently utilize their database run the risk of not measuring up to their competitors. Due to this increased importance, database management systems must be as helpful, logical and semantic as possible to users. The recommended solutions to the problems detailed in the previous chapter have all aimed to increase a system's semantic weaknesses. Query formulation is also aided by the recommendations.

The MENU and VIEW commands better familiarize end-users with the relations which, in total, create the database. Any questions concerning any relations may quickly be resolved. A set-type attribute allows relations to more accurately model the real world. All the facts are contained in a tuple and storage space is saved while redundancy is reduced. Query formulation is aided as users are no longer forced to think in terms of the physical database structure. Query language modifications allow individual values of set-type attributes to be selected. Joins are made more

semantically clear through an interactive dialogue between the system and user. User declared data types provide greater database integrity and flexibility.

These extensions to a relational DBMS serve to enhance a system's semantics and directly aide in query formulation. This in turn provides for a user-friendly system which will promote accuracy and efficiency while using one of the organizations most valued resources, its database.

LIST OF REFERENCES

1. Deen, S.M., Fundamentals of Data Base Systems, Hayden Book Company, December 1977.

2. Kroenke, D.M., Database Processing: Fundamentals, Design, Implementation, 2nd edition, SRA, 1983.

3. Wiorkowski, G., "Is 'Relational' Just a New Buzzword", Computerworld, Volume 17, No. 14, April 4, 1983, pp. 51, 56.

4. Date, C.J., An Introduction to Database Systems, 3rd edition, Addison Wesley, 1981.

5. Codd, E.F., "Relational Database: A Practical Foundation for Productivity", The 1987 ACM Turing Award Lecture, Communications of the ACM, Volume 25, No. 2, February 1982.

6. Salzberg, B.J., An Introduction to Data Base Design, Academic Press College Division, 1986.

7. Martorelli, W.P., "Relational DBMS Is On The Way to DP Dominance", Information Systems News, December 12, 1983, pp. 24, 26.

8. Kent, William, "Limitations of Record-Based Information Models", ACM Transactions on Database Systems, Volume 4, No. 1, March 1979.

9. Codd, E.F., "A Relational Model of Data for Large Shared Data Banks", Communications of the ACM, Volume 13, No. 6, June 1970.

10. Tsur, S., Zaniolo, C., "An Implementation of GEM-Supporting a Semantic Data Model on a Relational Backend.", Communications of the ACM 0-89791-128-8/84/006/0286 1984.

11. Wu, C.T., "Adding A Set-Type to a Relational System", unpublished paper, Naval Postgraduate School, Monterey, California, April 1988.

12. Gillin, Paul, "Relational Data Base Management Systems-Pushing for Acceptance", Computerworld, Volume 17, No. 52, December 26, 1983.

13. Brodie, Michael, L. and others, "On the Development of Data Models", On Conceptual Modelling, Springer-Verlag, 1984.

14. Sandberg, G. "A Primer or Relational Data Base Concepts", IBM Systems Journal, Volume 20, No. 1, 1981.

15. Elbinger, Lee, "Some Tips for Purchasing a Relational DBMS", Computerworld, Volume 17, No. 44, October 31, 1983.

INITIAL DISTRIBUTION LIST

		NO. PAGES
1.	Defense Technical Information Center Cameron Station Alexandria, Virginia 22304-6145	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93943-5002	2
3.	Chief of Naval Operations Director, Information Systems (OP-945) Navy Department Washington, D.C. 20350-2000	1
4.	Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93943-5000	2
5.	Curriculum Officer, Code 37 Computer Technology Naval Postgraduate School Monterey, California 93943-5000	1
6.	Professor C. Thomas Wu, Code 52Wq Computer Science Department Naval Postgraduate School Monterey, California 93943-5100	5
7.	Lt. Jon Wall PATWING ONE DET CUBI NAS Box 54 FPO San Francisco, California 96654-2906	2
8.	Mr. and Mrs. William A. Wall 2726 SW English Lane Portland, Oregon 97201	2